RCA PF020079 Ref AF

CITED BY APPLICANT

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 February 2001 (08.02.2001)

PCT

(10) International Publication Number
WO 01/09715 A2

(51) International Patent Classification[7]: G06F 9/00

(21) International Application Number: PCT/US00/20244

(22) International Filing Date: 25 July 2000 (25.07.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
09/363,283 28 July 1999 (28.07.1999) US

(71) Applicant: SUN MICROSYSTEMS, INC. [US/US]; 901 San Antonio Road, M/S: UPAL01-521, Palo Alto, CA 94303 (US).

(72) Inventors: UNGAR, David; 844 Driftwood Drive, Palo Alto, CA 94303 (US). WOLCZKO, Mario; 36 Coronado Avenue, San Carlos, CA 94070 (US).

(74) Agents: HECKER, Gary, A. et al.; The Hecker Law Group, Suite 2300, 1925 Century Park East, Los Angeles, CA 90067 (US).

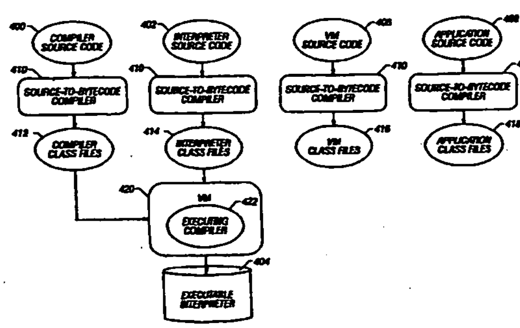
(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *Without international search report and to be republished upon receipt of that report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: A SINGLE-COMPILER ARCHITECTURE



(57) Abstract: Embodiments of the invention implement a single-compiler architecture configured to use one more compiler(s) to compile a kernel of a virtual machine (VM) that can be used to execute the compiler(s) as well as other program code that executes within the VM. According to one or more embodiments of the invention, in the single-compiler architecture, each compiler that is used supports similar operating conventions and/or a common run-time environment or architecture. In an embodiment of the invention, the VM initially comprises an executable interpreter and those portions of the VM needed to interpret program code (i.e., the VM kernel). The executable interpreter is generated using one or more compiler(s) executing in a VM. In an embodiment of the invention, the interpreter, the compiler(s), VM, and application programs that execute within the VM are written using the same programming language (e.g., the Java programming language). In embodiments of the invention, the interpreter determines when program code is needed for execution and whether the program code is to be interpreted or compiled. The executable interpreter is configured to add the compiler(s) and a memory system to the VM kernel as needed for program execution.

WO 01/09715 A2

A SINGLE-COMPILER ARCHITECTURE

BACKGROUND OF THE INVENTION

1. FIELD OF THE INVENTION

This invention relates to a computer system, and more specifically to a compiler architecture for use by a computer system.

2. BACKGROUND ART

A computer program is typically written in a high-level language that is translated into machine-readable instructions (i.e., instructions that are executable by a computer's processor) for a given computer processor. Compilers are used to translate high-level program code into machine-readable instructions and are written for specific hardware platforms (i.e., specific microprocessors). A program that is compiled to run on one hardware platform cannot run on another unless it is recompiled. However, it is possible to emulate

a first hardware platform on a second hardware platform such that the first platform's program can be run on the second platform without recompiling the program code. Such an emulation can be implemented as software that runs on the second platform and can be referred to as a virtual machine.

Problems arise when the virtual machine is written in one language and the program that is being run in the virtual machine is written in another language. Where different programming languages are used, for example, different compilers must be used to compile the program code. Incompatibilities between compilers raise issues that affect program design and execution. Each compiler typically uses different techniques to reference allocated memory making it difficult to determine whether the allocated memory can be reclaimed (a process referred to as garbage collection). Also, in a multithread environment (e.g., an environment capable of executing multiple units or threads of execution), it may be difficult to implement cooperative thread scheduling where programs are written in different programming languages. Thus, it would be beneficial to use a single programming language's compiler to generate both the virtual machine and the programs that execute in the virtual machine.

One example of a virtual machine is the Java virtual machine. The Java virtual machine interprets or translates bytecodes generated from programs that are written in the Java programming language into machine-level instructions that can be executed by the hardware platform. The Java virtual machine has been written using either the C or C++ programming languages. The C++ and Java programming languages are object-oriented programming languages.

The problems associated with using different programming languages' compilers for a virtual machine and the programs that run in the virtual machine

can be better understood from a review of a virtual machine's processing environment and an overview of object-oriented programming.

Object-Oriented Programming

Object-oriented programming is a method of creating computer programs by combining certain fundamental building blocks, and creating relationships among and between the building blocks. The building blocks in object-oriented programming systems are called "objects." An object is a programming unit that groups together a data structure (one or more instance variables) and the operations (methods) that can use or affect that data. Thus, an object consists of data and one or more operations or procedures that can be performed on that data. The joining of data and operations into a unitary building block is called "encapsulation."

An object can be instructed to perform one of its methods when it receives a "message." A message is a command or instruction sent to the object to execute a certain method. A message consists of a method selection (e.g., method name) and a plurality of arguments. A message tells the receiving object what operations to perform.

One advantage of object-oriented programming is the way in which methods are invoked. When a message is sent to an object, it is not necessary for the message to instruct the object how to perform a certain method. It is only necessary to request that the object execute the method. This greatly simplifies program development.

Object-oriented programming languages are predominantly based on a "class" scheme. The class-based object-oriented programming scheme is generally described in Lieberman, "Using Prototypical Objects to Implement

Shared Behavior in Object-Oriented Systems," OOPSLA 86 Proceedings, September 1986, pp. 214-223.

A class defines a type of object that typically includes both variables and methods for the class. An object class is used to create a particular instance of an object. An instance of an object class includes the variables and methods defined for the class. Multiple instances of the same class can be created from an object class. Each instance that is created from the object class is said to be of the same type or class.

To illustrate, an employee object class can include "name" and "salary" instance variables and a "set_salary" method. Instances of the employee object class can be created, or instantiated for each employee in an organization. Each object instance is said to be of type "employee." Each employee object instance includes "name" and "salary" instance variables and the "set_salary" method. The values associated with the "name" and "salary" variables in each employee object instance contain the name and salary of an employee in the organization. A message can be sent to an employee's employee object instance to invoke the "set_salary" method to modify the employee's salary (i.e., the value associated with the "salary" variable in the employee's employee object).

A hierarchy of classes can be defined such that an object class definition has one or more subclasses. A subclass inherits its parent's (and grandparent's etc.) definition. Each subclass in the hierarchy may add to or modify the behavior specified by its parent class. Some object-oriented programming languages support multiple inheritance where a subclass may inherit a class definition from more than one parent class. Other programming languages support only single inheritance, where a subclass is limited to inheriting the class definition of only one parent class. The Java programming language also

provides a mechanism known as an "interface" which comprises a set of constant and abstract method declarations. An object class can implement the abstract methods defined in an interface. Both single and multiple inheritance are available to an interface. That is, an interface can inherit an interface definition from more than one parent interface.

An object is a generic term that is used in the object-oriented programming environment to refer to a module that contains related code and variables. A software application can be written using an object-oriented programming language whereby the program's functionality is implemented using objects.

Platform-Independent Programming Languages and Program Execution

Object-oriented software applications typically comprise one or more object classes and interfaces. Many programming languages can be used to write a program that is compiled into machine-dependent (or platform-dependent) executable program code. However, in other languages such as the Java programming language, programs are compiled into platform-independent bytecode class files. Each class contains code and data in a platform-independent format. A bytecode includes a code that identifies an instruction (an opcode) and none or more operands to be used in executing the instruction. The computer system acting as the execution vehicle contains a program called a virtual machine, which is responsible for executing the platform-independent code (i.e., bytecodes generated from a program written using the Java programming language).

Applications may be designed as standalone applications, or as "applets" which are identified by an applet tag in an HTML (Hypertext Markup Language) document, and loaded by a browser application. The class files associated with

an application or applet may be stored on the local computing system, or on a server accessible over a network. Each class file is loaded into the virtual machine, as needed, by the "class loader."

To provide a client with access to class files from a server on a network, a web server application is executed on the server to respond to HTTP (Hypertext Transport Protocol) requests containing URLs (Universal Resource Locators) to HTML documents, also referred to as "web pages." When a browser application executing on a client platform receives an HTML document (e.g., as a result of requesting an HTML document by forwarding a URL to the web server), the browser application parses the HTML and automatically initiates the download of the specified bytecode class files when it encounters an applet tag in the HTML document.

The classes of an applet are loaded on demand from the network (stored on a server), or from a local file system, when first referenced during the applet's execution. The virtual machine locates and loads each class file, parses the class file format, allocates memory for the class's various components, and links the class with other already loaded classes. This process makes the code in the class readily executable by the virtual machine. Native code, e.g., in the form of a linked library, is loaded when a class file containing the associated native method is instantiated within the virtual machine.

Figure 1 illustrates the compile and runtime environments for a processing system. In the compile environment, a software developer creates class source files 100 which contain the programmer readable class definitions, including data structures, method implementations and references to other classes. Class source files 100 are provided to compiler 101, which compiles class source files 100 into compiled ".class" (or class) files 102 that contain bytecodes

executable by a virtual machine. Class files 102 are stored (e.g., in temporary or permanent storage) on a server, and are available for download over a network. Alternatively, class files 102 may be stored locally in a directory on the client platform.

The runtime environment contains virtual machine (VM) 105 which is able to execute bytecode class files and execute native operating system ("O/S") calls to operating system 109 when necessary during execution. Virtual machine 105 provides a level of abstraction between the machine independence of the bytecode classes and the machine-dependent instruction set of the underlying computer hardware 110, as well as the platform-dependent calls of operating system 109.

Class loader and bytecode verifier ("class loader") 103 is responsible for loading bytecode class files 102 and supporting class libraries 104 into virtual machine 105 as needed. Class loader 103 also verifies the bytecodes of each class file to maintain proper execution and enforcement of security rules. Within the context of runtime system 108, either an interpreter 106 executes the bytecodes directly, or a "just-in-time" (JIT) compiler 107 translates the bytecodes into machine code, so that they can be executed by the processor (or processors) in hardware 110.

Native code, e.g., in the form of a linked library 111, is loaded when a class (e.g., from class libraries 104) containing the associated native method is instantiated within the virtual machine. Linked library 111 can be, for example, a "shared object" library in the Solaris™ or UNIX environment that is written as a ".so" file, or linked library 111 may take the form of a dynamic loadable library written as a ".dll" file in a Windows environment.

Interpreter 106 reads, interprets and executes a bytecode instruction before continuing on to the next instruction. JIT compiler 107 can translate multiple bytecode instructions into machine code that are then executed. Compiling the bytecodes prior to execution results in faster execution. If, for example, the same bytecode instruction is executed multiple times in a program's execution, it must be interpreted each time it is executed using interpreter 106. If JIT compiler 107 is used to compile the program, the bytecode instruction may be translated once regardless of the number of times it is executed in the program. Further, if the compilation (i.e., output of JIT compiler 107) is retained, there is no need to translate each instruction during program execution.

The runtime system 108 of virtual machine 105 supports a general stack architecture. The manner in which this general stack architecture is supported by the underlying hardware 110 is determined by the particular virtual machine implementation, and reflected in the way the bytecodes are interpreted or JIT-compiled. Other elements of the runtime system include thread management (e.g., scheduling) and garbage collection mechanisms.

Figure 2 illustrates runtime data areas which support the stack architecture within runtime system 108. In Figure 2, runtime data areas 200 comprise one or more thread-based data areas 207. Each thread-based data area 207 comprises a program counter register (PC REG) 208, a local variables pointer register (VARS REG) 209, a frame register (FRAME REG) 210, an operand stack pointer register (OPTOP REG) 211, a stack 212 and, optionally, a native method stack 216. Stack 212 comprises one or more frames 213 which contain an operand stack 214 and local variables 215. Native method stack 216 comprises one or more native method frames 217.

Runtime data areas 200 further comprise shared heap 201. Heap 201 is the runtime data area from which memory for all class instances and arrays is allocated. Shared heap 201 comprises method area 202, which is shared among all threads. Method area 202 comprises one or more class-based data areas 203 for storing information extracted from each loaded class file. For example, class-based data area 203 may comprise class structures such as constant pool 204, field and method data 205, and code for methods and constructors 206.

A virtual machine can support many threads of execution at once. Each thread has its own thread-based data area 207. At any point, each thread is executing the code of a single method, the "current method" for that thread. If the "current method" is not a native method, program counter register 208 contains the address of the virtual machine instruction currently being executed. If the "current method" is a native method, the value of program counter register 208 is undefined. Frame register 210 points to the location of the current method in method area 202.

Each thread has a private stack 212, created at the same time as the thread. Stack 212 stores one or more frames 213 associated with methods invoked by the thread. Frames 213 are used to store data and partial results, as well as to perform dynamic linking, return values for methods and dispatch exceptions. A new frame is created and pushed onto the stack each time a method is invoked, and an existing frame is popped from the stack and destroyed when its method completes. A frame that is created by a thread is local to that thread and typically cannot be directly referenced by any other thread.

Each frame 213 has its own set of local variables 215 and its own operand stack 214. The local variables pointer register 209 contains a pointer to the base

of an array of words containing local variables 215 of the current frame. The operand stack pointer register 211 points to the top of operand stack 214 of the current frame. Most virtual machine instructions take values from the operand stack of the current frame, operate on them, and return results to the same operand stack. Operand stack 214 is also used to pass arguments to methods and receive method results.

Native method stack 216 stores native method frames 217 in support of native methods. Each native method frame provides a mechanism for thread execution control, method arguments and method results to be passed between methods and native methods implemented as native code functions in a linked library.

Garbage Collection

A garbage collection mechanism is typically used to reclaim memory allocated to objects that are no longer needed. If, for example, an instance variable of one object points to, or references, a second object, the second object is still being used and the memory associated with the second object cannot be reclaimed. However, once there are no more references to the second object, its memory can be reclaimed. In virtual machine 105, a garbage collector can examine operand stack 214 and local variables stack 215 to locate pointers to objects. Thus, there are identifiable locations that contain pointers to objects and can be examined by the garbage collector. In the C++ programming language, however, there is no mechanism to allow the garbage collector to determine where the C++ compiler has stored local variables that contain pointers to objects. Since there is no guarantee that a local variable contains a pointer to an object, the C++ compiler cannot flag a local variable as containing a pointer to an object. Thus, it is difficult to determine whether or not a pointer exists to an

object. Thus, it is difficult to determine what memory is available for reclamation in a C++ program. Since virtual machine 105 is written in C++, it is difficult to perform garbage collection on virtual machine 105.

To compensate for the lack of pointer information provided by the C++ compiler, C++ developers use a special C++ object referred to as a handle. A pointer to a C++ object is embedded in the handle object. The handle object may be queried to determine an object pointer. Further, the handle registers with a catalog that can be referenced by the garbage collector to locate all handles, and thus all object pointers embedded in handle objects. This technique of tracking pointers adds overhead to the program execution and development. Since VM 105 is written in C++, to accommodate garbage collection, it must be written with the added overhead associated with using handle objects to store object pointers. During execution, memory must be allocated for each handle object. Further, it is necessary to query the handle object to retrieve an object pointer before the pointer can be used to reference the object. This old approach can lead to bugs because the code must explicitly free the handles.

<u>Thread Scheduling</u>

In a multithread environment, threads of execution (a unit of operation that is scheduled and to which resources such as execution time, locks and queues can be assigned) can be scheduled so that resources can be optimized, for example. A thread may be created when a method of a class is invoked, for example. A cooperative scheduling technique typically requires that a thread be able to yield to another thread. For example, a loop that is executed in one thread can contain a "yield" operation that would allow the loop execution to stop so that another thread can be executed.

Unlike other compilers (e.g., Java compilers), C++ compilers do not provide a mechanism for automatically inserting yield operations. Thus, thread scheduling is either left to each program developer to manually insert yield operations into a program, or non-cooperative scheduling is used. If non-cooperative scheduling is used, each program must be written to accommodate the possibility that it may be stopped during execution to allow another program's thread to execute.

Existing virtual machine approaches use multiple compilers such that different compilers are used to compile the virtual machine program code and the program code that is executed in the virtual machine. Since multiple compilers are used, there is no way of ensuring that uniform facilities are available to provide such functionality as garbage collection and cooperative scheduling, for example.

SUMMARY OF THE INVENTION

Embodiments of the invention comprise a single-compiler architecture. According to one or more embodiments of the invention, a compiler is executed in a virtual machine (VM) to generate an executable version of an interpreter (or executable interpreter). In an embodiment of the invention, the executable interpreter is configured to interpret program code for execution on the computer system, determine when program code is needed and whether it is to be interpreted or compiled, and maintain usage statistics.

The executable interpreter provides a kernel of a VM that can be used as a bootstrapping mechanism to load additional functionality of a VM. The executable interpreter can interpret the class files of the compiler used to generate the executable interpreter as well as other compiler's class files to add at least one compiler's functionality to the VM. Therefore, program code to be executed in the VM can be either compiled or interpreted. In addition, to compiler program code, the VM can either compile or interpret VM program code (e.g., a memory manager) as well as application program code.

According to one or more embodiments of the invention, the compiler(s) and interpreter are written using the Java programming language. A compiler is executed in a statically-compiling mode within a VM to generate an executable interpreter that includes an interpreter and any portions of the VM that may be called by the interpreter to interpret bytecodes.

When executed, the executable interpreter provides a VM kernel with a bootstrapping mechanism for invoking the compiler(s) and then incorporating the resultant machine-executable code into the VM. In an embodiment of the invention, the executable interpreter determines when program code is to be interpreted or compiled. For example, if the executable interpreter determines

that portions of the compiler, memory management code, or application code are being executed frequently, it will invoke at least one of the VM's compilers on those portions and incorporate the resultant compiled code into the VM. In this manner, code that is executed frequently will be executed in compiled form.

According to one or more embodiments of the invention the same compiler(s) used to generate the executable version of the interpreter may be used in the VM to compile program code. Each compiler that is used supports similar operating conventions and/or a common run-time environment or architecture. Thus, there can be interpreted and compiled portions of the virtual machine as well as the application programs that execute in the virtual machine. There can be interpreted and compiled portions of the compiler as well.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a block diagram of compile and runtime environments.

Figure 2 is a block diagram of the runtime data areas of an embodiment of a virtual machine.

Figure 3 is a block diagram of one embodiment of a computer system capable of providing a suitable execution environment for an embodiment of the invention.

Figures 4A-4C provides an overview of the single-compiler architecture according to an embodiment of the invention.

Figure 5 provides an execution process according to an embodiment of the invention.

Figure 6 illustrates usage of a single compiler to compile all program code according to an embodiment of the invention.

The Figure 7 provides an interpreter generation process according to an embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

A single-compiler architecture is described. In the following description, numerous specific details are set forth in order to provide a more thorough description of the present invention. It will be apparent, however, to one skilled in the art, that the present invention may be practiced without these specific details. In other instances, well-known features have not been described in detail so as not to obscure the invention.

Though discussed herein with respect to the Java programming language and the Java virtual machine (JVM), the invention is not limited to the Java programming language and/or the Java virtual machine.

Embodiments of the invention implement a single-compiler architecture in which program code of a VM and the program code that executes in the VM are compiled using one or more compilers that support similar operating conventions and/or a common run-time environment or architecture. For example, a program code of the VM and program code running in a VM may be compiled using one or more compilers that support common garbage collection, thread scheduling and/or memory management mechanism(s).

Initially, an interpreter and bimodal compiler are written. The interpreter includes the capability to identify frequently executed portions of code, to invoke one or more compilers to compile such code, to incorporate the resultant machine code into the virtual machine, and to subsequently mix execution of interpreted and compiled code. The interpreter is comprised of the interpreter itself and any routines that are called by the interpreter to interpret an interpretable form of data. Two compiler modes that are used in embodiments of the invention are static and dynamic. In static mode, a compiler compiles all of a program at once, in dynamic mode it compiles pieces of a program on an as-

needed basis. When the interpreter invokes the compiler, it uses the dynamic mode. When the compiler compiles the interpreter as described below, it uses the static mode. In an embodiment of the invention, the interpreter and compiler may be written in the Java programming language that together comprise a high-performance VM.

Once the interpreter and bimodal compiler are written, a preexisting source-to-interpretable-form compiler to bytecodes converts the components from source code to an interpretable form. In an embodiment of the invention, the javac compiler is used, and the interpretable form consists of Java class files. Then, a preexisting virtual machine is used to execute the compiler (in a statically compiling mode). The compiler is given the interpretable-form interpreter as its input and produces a machine-executable version of the interpreter. In an embodiment of the invention, the preexisting virtual machine could be the VM, which would execute the compiler (as class files). The compiler would read the interpreter (as class files) and produce a machine-executable file.

Next, the machine-executable version of the interpreter is run with input consisting of the interpretable-form compiler and the user's application, also in interpretable form. As it runs, it identifies portions of code to be compiled. An example of a compiling criterion is execution frequency, but one familiar in the art will understand that other criteria may be used as well. These portions may belong to the compiler, to the user's application, or to ancillary functions, such as the storage management system. When such a portion is identified, the interpreter invokes a compiler to create a more-efficient, machine-executable representation of the portion. The machine-executable code is then incorporated into the virtual machine so that frequently executed code is executed directly as machine code and infrequently executed code is interpreted. In an embodiment

of the invention, the machine-executable version of the interpreter constitutes a VM, and the interpretable form is represented as Java class files.

In embodiments of this invention, the same compiler is used to compile both the virtual machine including the interpreter, and portions of the user's application. In other embodiments of the invention, more than one compiler is used wherein each of compilers support similar operating conventions and/or a run-time architectures. In turn, this results in simpler support for garbage collection, and multithreading. It also results in a smaller memory footprint for the virtual machine, and makes it easier to port the virtual machine to another platform.

Embodiment of Computer Execution Environment (Hardware)

An embodiment of the invention can be implemented as computer software in the form of computer readable code executed on a general purpose computer such as computer 300 illustrated in Figure 3, or in the form of bytecode class files executable within a runtime environment running on such a computer, or in the form of bytecodes running on a processor (or devices enabled to process bytecodes) existing in a distributed environment (e.g., one or more processors on a network). A keyboard 310 and mouse 311 are coupled to a system bus 318. The keyboard and mouse are for introducing user input to the computer system and communicating that user input to processor 313. Other suitable input devices may be used in addition to, or in place of, the mouse 311 and keyboard 310. I/O (input/output) unit 319 coupled to system bus 318 represents such I/O elements as a printer, A/V (audio/video) I/O, etc.

Computer 300 includes a video memory 314, main memory 315 and mass storage 312, all coupled to system bus 318 along with keyboard 310, mouse 311 and processor 313. The mass storage 312 may include both fixed and removable

media, such as magnetic, optical or magnetic optical storage systems or any other available mass storage technology. Bus 318 may contain, for example, thirty-two address lines for addressing video memory 314 or main memory 315. The system bus 318 also includes, for example, a 64-bit data bus for transferring data between and among the components, such as processor 313, main memory 315, video memory 314 and mass storage 312. Alternatively, multiplex data/address lines may be used instead of separate data and address lines.

In one embodiment of the invention, the processor 313 is a SPARC microprocessor from Sun Microsystems, Inc., a microprocessor manufactured by Motorola, such as the 680X0 processor or a microprocessor manufactured by Intel, such as the 80X86, or Pentium processor. However, any other suitable microprocessor or microcomputer may be utilized. Main memory 315 is comprised of dynamic random access memory (DRAM). Video memory 314 is a dual-ported video random access memory. One port of the video memory 314 is coupled to video amplifier 316. The video amplifier 316 is used to drive the cathode ray tube (CRT) raster monitor 317. Video amplifier 316 is well known in the art and may be implemented by any suitable apparatus. This circuitry converts pixel data stored in video memory 314 to a raster signal suitable for use by monitor 317. Monitor 317 is a type of monitor suitable for displaying graphic images. Alternatively, the video memory could be used to drive a flat panel or liquid crystal display (LCD), or any other suitable data presentation device.

Computer 300 may also include a communication interface 320 coupled to bus 318. Communication interface 320 provides a two-way data communication coupling via a network link 321 to a local network 322. For example, if communication interface 320 is an integrated services digital network (ISDN) card or a modem, communication interface 320 provides a data communication connection to the corresponding type of telephone line, which comprises part of

network link 321. If communication interface 320 is a local area network (LAN) card, communication interface 320 provides a data communication connection via network link 321 to a compatible LAN. Communication interface 320 could also be a cable modem or wireless interface. In any such implementation, communication interface 320 sends and receives electrical, electromagnetic or optical signals which carry digital data streams representing various types of information.

Network link 321 typically provides data communication through one or more networks to other data devices. For example, network link 321 may provide a connection through local network 322 to local server computer 323 or to data equipment operated by an Internet Service Provider (ISP) 324. ISP 324 in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" 325. Local network 322 and Internet 325 both use electrical, electromagnetic or optical signals which carry digital data streams. The signals through the various networks and the signals on network link 321 and through communication interface 320, which carry the digital data to and from computer 300, are exemplary forms of carrier waves transporting the information.

Computer 300 can send messages and receive data, including program code, through the network(s), network link 321, and communication interface 320. In the Internet example, remote server computer 326 might transmit a requested code for an application program through Internet 325, ISP 324, local network 322 and communication interface 320.

The received code may be executed by processor 313 as it is received, and/or stored in mass storage 312, or other non-volatile storage for later execution. In this manner, computer 300 may obtain application code in the

form of a carrier wave. In accordance with an embodiment of the invention, examples of such downloaded applications include one or more elements of a runtime environment, such as the virtual machine, class loader, class bytecode files, class libraries and the apparatus that implements the single-compiler architecture described herein.

Application code may be embodied in any form of computer program product. A computer program product comprises a medium configured to store or transport computer readable code or data, or in which computer readable code or data may be embedded. Some examples of computer program products are CD-ROM disks, ROM cards, floppy disks, magnetic tapes, computer hard drives, servers on a network, and carrier waves.

The computer systems described above are for purposes of example only. An embodiment of the invention may be implemented in any type of computer system or programming or processing environment, including embedded devices (e.g., web phones, etc.) and "thin" client processing environments (e.g., network computers (NC's), etc.) that support a virtual machine.

Single-Compiler Architecture

According to one or more embodiments of the invention, similar operating conventions and/or a common run-time environment or architecture are supported by each of the compilers used to compile a kernel VM as well as input program code for a kernel VM. The kernel VM can be used to execute the same (or additional) compiler(s) to compile its own program code, the program code of other portions of the VM, and/or application program code. Initially, the kernel VM consists of an executable interpreter that includes an interpreter and any routines otherwise external to the interpreter that may be needed to interpret bytecodes. The executable interpreter determines whether other

portions of the VM are needed to execute program code. Those other portions are then retrieved as input program code for the VM. It may be necessary, for example, to execute one or more of the compilers to compile program code or to execute the memory system to perform a memory management function.

When program code is needed for execution, it is either interpreted or compiled by the VM in what is called "mixed mode" execution. Mixed mode execution refers to the use of two or more execution mechanisms, e.g., the interpreter and the compiler, to selectively execute portions of the input program code provided to the VM. The executable interpreter determines whether program code should be compiled and causes at least one compiler to be executed to compile the program code as needed. Compiled versions may be stored for later use. Thus, if a compiled version of the program code is to be used, a determination can be made whether a compiled version of the program code already exists. If so, the stored compiled version of the program code is retrieved and executed (e.g., by branching to the compiled code). If there is no stored compilation of the program code, program code is input to at least one compiler for compilation. The compiled version of program code can be executed in place of an interpreted version of program code. The compilation process may be performed, for example, by interrupting the interpreter to compile within the current thread of execution, or by spawning a new thread of execution in which the compilation process is carried out.

The single-compiler architecture according to an embodiment of the invention is illustrated in Figures 4A-4C. As is discussed below, the single-compiler architecture in Figures 4A-4C may comprise multiple instances of compiler 432 and their respective class files 412. That is, compiler 432 and class files 412 may represent more than one compiler and compiler class files, respectively according to one or more embodiments of the invention.

In one or more embodiments of the invention, a first VM is used to generate a kernel VM or, the executable interpreter. The executable interpreter can then be used as a second VM that initially comprises the executable interpreter and may further comprise the compiler used to create the executable interpreter, and other VM program code functionality (e.g., memory system).

Figure 4A illustrates the generation of the executable interpreter according to an embodiment of the invention. Figure 4B illustrates the use of an initial execution system, or kernel VM, that comprises an executable interpreter according to an embodiment of the invention. Figure 4C illustrates a VM comprising the executable interpreter, the compiler that generated the executable interpreter, and other VM program code according to an embodiment of the invention.

Referring to Figure 4A, compiler source code 400 may be written using the Java programming language. Interpreter source code 402, VM source code 406 and application source code may also be written using the Java programming language. A source-to-bytecode compiler 410 is used to generate compiler class files 412 and interpreter class files 414. Source-to-bytecode compiler 410 is further used to generate VM class files 416 and application class files 418.

Class files 412 are executed using VM 420 to generate the executable interpreter 404 (using class files 414). VM 420 includes functionality (e.g., an interpreter or compiler) to execute compiler class files 412 as executing compiler 422. Executing compiler 422 compiles interpreter class files 414 (and other VM routines, if needed) to generate executable interpreter 404. Executable interpreter 404 contains executable (e.g., machine-readable) program code.

Executable interpreter 404 provides a kernel VM, or an execution system, that initially comprises an interpreter and those portions of a VM that are called by the interpreter to interpret bytecodes. Some elements that may be included in the VM kernel are, for example, a class loader, a thread support element (e.g., providing scheduling, synchronization, etc.), or a garbage collector. The execution system can invoke the compiler to optimize additional code needed to execute application, or other, program code. Referring to Figure 4B, executable interpreter 404 can invoke the compiler to compile portions of the VM as needed to execute program code. The additional code can be, for example, a compiler to compile program code and a memory system to manage the memory needed for executing program code. The compiler that is added to the execution system is the same compiler that is used to generate executable interpreter 404 as illustrated with reference to Figure 4A.

Referring to Figure 4C, portions of compiler 432 can be interpreted by executable interpreter 404 from compiler class files 412. Further, compiler 432 can be used to compile its own class files (e.g., compiler class files 412). Thus, compiler 432 can be interpreted by executable interpreter 404, executed by compiler 432, or both. When compiler 432 is added to VM 430, program code (e.g., compiler class files 412, VM class files 416, memory system 442 and application class files 418) can be either compiled or interpreted.

In an embodiment of the invention, more than one instance of compiler 432 may be used to compile program code wherein the compiler instances support similar operating conventions and/or a common run-time environment or architecture. For example, each instance of compiler 432 may provide support for a garbage collection mechanism such that object pointers are stored in identifiable locations (e.g., operand stack 214 and local variables stack 215) that

can be examined by the garbage collector. Similarly, each instance of compiler 432 may provide support for a cooperative thread scheduling.

In an embodiment of the invention, executable interpreter 404 is configured to determine whether the program code to be executed should be compiled instead of interpreted. For example, executable interpreter 404 can determine when a given portion of program code is executed on a frequent basis. Such program code is compiled by compiler 432. In an embodiment of the invention, the compiled version of the program code can be stored for subsequent use.

VM 430 has been described as being compiled incrementally as needed (e.g., based on frequency of execution) during application program code execution. It should be apparent that portions of VM 430 can be compiled prior to executing the application program code.

To reduce the startup time, portions of the compiler (or other program code) can be pre-compiled (e.g., compiled prior to startup) in an embodiment of the invention. Journaling can be used in an embodiment of the invention to save intermediate compilation states of, for example, VM 430. Similarly, subsequent executions of an application can make use of previously compiled segments, or portions, of the application.

Garbage Collection

In the single-compiler architecture of one or more embodiments of the invention, VM class files 416, application class files 418, compiler class files 412 and executable interpreter 404 may be written using the Java programming language in embodiments of the invention. Thus, the same runtime environment can be used to run executable interpreter 404, compiler 432 and

memory system 442 (i.e., VM 430 of Figure 4C) as well as application class files 418.

Use of the same runtime environment facilitates garbage collection. In the single-compiler architecture, each compiler can be written to communicate with a garbage collection mechanism, or garbage collector, to provide garbage collection information, such as a list of pointers to objects at each garbage collection point (gc point). Thus, the garbage collector is able to collect unused memory previously allocated to VM 430 and the program code executed by VM 430.

Referring to Figure 2, for example, stack 212 contains all of the pointers to objects referenced by VM 430 (e.g., executable interpreter 404, compiler 432, memory system 442) and applications that run in VM 430. A garbage collection mechanism scans stack 212 to locate object pointers so that unreferenced objects can be identified and resources allocated to unreferenced objects can be reclaimed.

<u>Thread Scheduling</u>

The single-compiler architecture in embodiments of the invention facilitates scheduling in a multithread environment. For example, yield operations can be inserted into the program code (e.g., in loops) to facilitate cooperative scheduling of threads in a multithread environment.

In embodiments of the invention, application class files 418 comprise class files (having bytecode instructions) written using the Java programming language. It is possible, however, that application class files 418 can comprise code that is not written using the Java programming language (e.g., program code written using the C or C++ programming language). Such code may be

used to perform functions otherwise not supported, such as interfacing with specialized hardware (e.g., display hardware) or software (e.g., database drivers) of a given platform, or may also be used to speed up computationally intensive functions, such as rendering. It is possible for class files written using the Java programming language to interoperate with other program code (i.e., native code or machine-dependent program code). For example, the native code can be placed in a different process using inter-process calls such that the process that executes class files that are written using the Java programming language and the process that executes native code can communicate.

Native code may block other thread execution thereby undermining cooperative scheduling of threads. A technique referred to as binary translation can be used to facilitate interoperability such that native code is imported for execution using a VM. When native code is to be executed within a virtual machine environment, the native code is translated into an intermediate form. The intermediate form of the native code is compiled or interpreted by the virtual machine to execute the routines defined by the native code. Because the revised native code is not allowed to block other threads, thread scheduling may be performed by the virtual machine rather than the underlying operating system, and cooperative scheduling may be performed. A more detailed discussion of binary translation provided in a co-pending patent application entitled "Method and Apparatus for Translating and Executing Native Code in a Virtual Machine Environment", patent application serial number 09/134,073, filed on August 13, 1998 and assigned to a common assignee is incorporated herein by reference.

Execution Process Flow

Program code is executed in VM 430 using executable interpreter 404 and/or compiler 432. Figure 5 provides a process for execution of program code according to an embodiment of the invention. An executable version of the interpreter (e.g., executable interpreter 404) is generated at step 502. At step 504, executable interpreter 404 is loaded in a computer system, or target machine. Executable interpreter 404 is used to interpret program code for execution on, for example, the target system. At step 506, a determination is made by executable interpreter 404 whether to interpret the program code or compile the program code. Executable interpreter 404 can determine whether to use a compiled or interpreted version of program code based on the number of times the program code is executed, for example.

If executable interpreter 404 determines that the program code is to be interpreted, processing continues at step 508. Usage statistics (e.g., the number of times or frequency that a given portion of program code is executed) can be maintained at step 508 such that executable interpreter 404 can determine (at step 506) whether to interpret the program code or cause it to be compiled by compiler 432. At step 510, executable interpreter 404 interprets the program code such that, at step 518, the interpreted version of the program code is executed on the target machine or computer system.

If it is determined (at step 506) that the program code should be compiled, processing continues at step 512. At step 512, a determination is made whether a compiled version of the program code already exists. If so, processing continues at step 518 to execute the compiled version of the program code using hardware 416.

If it is determined (at step 512) that a compiled version of the program code does not exist, processing continues at step 514 to compile the program code (e.g., using compiler 432). At step 516, the results of the compilation of the program code by compiler 432 are retained by the VM (e.g., VM 430) for subsequent execution, if necessary. Alternatively, the compilation results can be discarded after execution. This may be beneficial where the computer system has limited storage capacity, for example. At step 518, the compiled version of the program code is executed on, for example, a target machine or computer system.

In one or more embodiments of the invention, program code that is compiled by compiler 432 (as well as additional compiler instances that support similar operating conventions and/or a common run-time environment or architecture) at step 514 can be compiler class files 412, application class files 418 and/or VM class files 416, for example. Thus, executable interpreter 404, the compiler itself, VM program code and application program code may be compiled by one or more compilers that adopt similar operating conventions or a common runtime environment or architecture. Figure 6 illustrates the use of a single-compiler architecture to compile program code according to an embodiment of the invention.

To execute compiler program code, compiler class files 412 (n.b., compiler class files 412 may represent one or more compiler's class files) can be input to executable interpreter 404 to implement interpretive execution of a portion of compiler 432 (where compiler 432 may represent than one compiler instance), with interpreted compiler 606A being the resulting effect of that execution. Interpreted compiler 606A can be used to compile program code. Where the program code to be compiled is the compiler itself, compiler class files 412 can be input to interpreted compiler 606A to generate compiled compiler 606B.

Compiler class files 412 can also be input to compiled compiler 606B to generate additional portions of compiled compiler 606B. In one or more embodiments of the invention, compiler 606B may be represent more than one compiler instance.

Similarly, VM class files 416 can be executed using executable interpreter 404, interpreted compiler 606A and/or compiled compiler 606B. For example, when memory allocation is needed for an application program, memory system 442 is translated to machine-readable code using executable interpreter 404, interpreted compiler 606A and/or compiled compiler 606B. If some or all of memory system 442 is executed with some frequency, interpreted compiler 606A and/or compiled compiler 606B can be executed to compile the frequently-executed portion(s) of memory system 442's program code.

If interpreted compiler 606A is used, compiler class files 412 is input to executable interpreter 404 to implement interpreted compiler 606A. The needed memory system program code is input to interpreted compiler 606A to generate a compiled version of the memory system program code. Similarly, where compiled compiler 606B is used, the memory system program code is input to compiled compiler 606B. If the portion of memory system 408's program code does not need to be compiled, the program code is input to executable interpreter 404.

Application class files 416 can be executed using executable interpreter 404, interpreted compiler 606A and/or compiled compiler 606B. If an application's class files are compiled, they can be compiled using interpreted compiler 606A and/or compiled compiler 606B. Otherwise, executable interpreter 404 is used to interpret application class files 418.

Interpreter Generation Process Flow

Executable interpreter 404 is comprised of machine-readable program code that can be executed on a computer system. In an embodiment of the invention, source program code for executable interpreter 404 that is written using the Java programming language is translated into Java programming language class file(s) that contain bytecode instructions. The interpreter class files are compiled to create a executable interpreter 404. According to an embodiment of the invention, the generation of executable interpreter 404 via a compilation process is performed using a compiler that is written using the Java programming language. The Figure 7 provides an interpreter generation process according to an embodiment of the invention.

At step 702, compiler 432's program code is input to a source-to-bytecode compiler to generate class files (e.g., compiler class files 412) for compiler 432. At step 704, executable interpreter 404's program code is input to a source-to-bytecode compiler to generate class files (e.g., interpreter class files 414) for executable interpreter 404.

At step 706, a bytecode compiler (i.e., one or more of the class files for compiler 432 that were generated in step 702) is executed in a VM. The bytecode compiler of step 706 that is executing in a VM is used to compile executable interpreter 404's class files at step 708. The output of step 708 is executable interpreter 404 at step 710. Executable interpreter 404 provides a standalone program that can be loaded on a computer system and executed as a kernel VM that can interpret class files (e.g., compiler class files 412, VM class files 416 and application class files 418) as previously described.

Thus, a single-compiler architecture has been described in conjunction with one or more specific embodiments. The invention is defined by the claims and their full scope of equivalents.

CLAIMS

What is claimed is:

1. A method for implementing a virtual machine in a computer system, comprising:

using a compiler to obtain a machine-executable version of a virtual machine kernel; and

running the virtual machine kernel to provide a first virtual machine configured to perform mixed-mode execution of input program code using an interpreter and the compiler; and

providing the compiler to the first virtual machine as part of the input program code;

wherein mixed-mode execution comprises determining whether a current portion of input program code is to be compiled or interpreted, and wherein the current portion of input program code comprises a portion of the compiler.

2. The method of claim 1, wherein determining whether the current portion of input program code is to be compiled or interpreted comprises determining an execution frequency of the current portion of input program code.

3. The method of claim 1, wherein mixed-mode execution further comprises:

compiling the current portion of input program code with the compiler to generate a corresponding compiled portion of code when the current portion of input program code is to be compiled; and

interpreting the current portion of input program code when the current portion of input program code is to be interpreted.

4. The method of claim 3, further comprising branching to the corresponding compiled portion of code for the current portion of input program code.

5. The method of claim 3, wherein compiling the current portion of input program code comprises interrupting the interpreter.

6. The method of claim 3, wherein compiling the current portion of input program code comprises spawning a separate thread of execution wherein the compiling is performed.

7. The method of claim 1, wherein using the compiler to obtain the machine executable version of the virtual machine kernel comprises executing the compiler in a second virtual machine.

8. The method of claim 1, wherein the input program code further comprises a memory system.

9. The method of claim 1, wherein the virtual machine kernel further comprises at least one of a garbage collector, a class loader and a thread support element.

10. The method of claim 1, wherein the compiler provides garbage collection information.

11. The method of claim 1, wherein:

the compiler operates in a static mode when obtaining the machine-executable version of the virtual machine kernel; and

the compiler operates in a dynamic mode when executing within the first virtual machine.

12. The method of claim 1, wherein the virtual machine kernel and the compiler are written in the same object-oriented programming language.

13. The method of claim 12, wherein the object-oriented programming language provides support for garbage collection.

14. The method of claim 13, further comprising converting the compiler and the virtual machine kernel from source code into virtual machine-readable bytecodes.

15. A computer program product comprising:

a computer usable medium having computer readable program code embodied therein for implementing a first virtual machine in a computer system, the computer readable program code obtained via a compiler, the computer readable program code comprising:

computer readable program code configured to cause the computer to receive the compiler as part of input program code;

computer readable program code configured to cause a computer to perform mixed-mode execution of the input program code using an interpreter and the compiler, wherein mixed-mode execution comprises determining whether a current portion of input program code is to be compiled or interpreted, wherein the current portion of input program code comprises a portion of the compiler.

16. The computer program product of claim 15, wherein the computer readable program code configured to cause the computer to perform mixed-mode execution further comprises computer readable program code configured to cause the computer to determine an execution frequency of the current portion of input program code.

17. The computer program product of claim 15, wherein the computer readable program code configured to cause the computer to perform mixed-mode execution further comprises:

computer readable program code configured to cause the computer to compile the current portion of input program code with the compiler to generate a corresponding compiled portion of code when the current portion of input program code is to be compiled; and

computer readable program code configured to cause the computer to interpret the current portion of input program code when the current portion of input program code is to be interpreted.

18. The computer program product of claim 17, further comprising computer readable program code configured to cause the computer to branch to the corresponding compiled portion of code for the current portion of input program code.

19. The computer program product of claim 17, wherein the computer readable program code configured to cause the computer to compile the current portion of input program code comprises computer readable program code configured to cause the computer to interrupt the interpreter.

20. The computer program product of claim 17, wherein the computer readable program code configured to cause the computer to compile the current portion of input program code comprises computer readable program code configured to cause the computer to spawn a separate thread of execution wherein the compiling is performed.

21. The computer program product of claim 15, wherein the computer readable program code configured to implement the first virtual machine is obtained via the compiler executing in a second virtual machine.

22. The computer program product of claim 15, further comprising computer readable program code configured to cause the computer to receive a memory system as part of input program code.

23. The computer program product of claim 15, further comprising at least one of a garbage collector, a class loader and a thread support element.

24. The computer program product of claim 15, wherein the compiler is configured to provide garbage collection information.

25. The computer program product of claim 15, wherein:

the compiler is configured to operate in a static mode when obtaining the computer readable program code configured to implement the first virtual machine; and

the compiler is configured to operate in a dynamic mode when executing within the first virtual machine.

26. The computer program product of claim 15, wherein the compiler and the computer readable program code configured to implement the first virtual machine are written in the same object-oriented programming language.

27. The computer program product of claim 26, wherein the object-oriented programming language provides support for garbage collection and synchronization.

28. The computer program product of claim 27, further comprising computer readable program code configured to cause a computer to convert the compiler and the computer readable program code configured to implement the first virtual machine from source code into virtual machine-readable bytecodes.

29. A single-compiler system comprising:

a compiler; and

a virtual machine kernel obtained using the compiler, the virtual machine kernel configured to perform mixed-mode execution of input program code using an interpreter and the compiler, wherein the input program code comprises the compiler;

wherein the virtual machine kernel is configured to determine whether a current portion of input program code is to be compiled or interpreted, and wherein the current portion of input program code comprises a portion of the compiler.

30. The system of claim 29, wherein the virtual machine kernel is configured to determine an execution frequency of the current portion of input program code.

31. The system of claim 29, wherein the virtual machine kernel is configured to:

compile the current portion of input program code with the compiler to generate a corresponding compiled portion of code when the current portion of input program code is to be compiled; and

interpret the current portion of input program code when the current portion of input program code is to be interpreted.

32. The system of claim 31, wherein the virtual machine kernel is configured to branch to the corresponding compiled portion of code for the current portion of input program code.

33. The system of claim 31, wherein the virtual machine kernel is configured to interrupt the interpreter to invoke the compiler.

34. The system of claim 31, wherein the virtual machine kernel is configured to spawn a separate thread of execution wherein the compiling is performed.

35. The system of claim 29, further comprising a virtual machine in which the compiler is executed to obtain the virtual machine kernel.

36. The system of claim 29, wherein the input program code further comprises a memory system.

37. The system of claim 29, wherein the virtual machine kernel further comprises at least one of a garbage collector, a class loader and a thread support element.

38. The system of claim 29, wherein the compiler provides garbage collection information.

39. The system of claim 29, wherein the compiler is configured to:

operate in a static mode when obtaining the virtual machine kernel; and

operate in a dynamic mode when executing within the virtual machine kernel.

40. The system of claim 29, wherein the virtual machine kernel and the compiler are written in the same object-oriented programming language.

41. The system of claim 40, wherein the object-oriented programming language provides support for garbage collection and synchronization.

42. The system of claim 41, wherein the compiler comprises bytecodes, and wherein the compiler is configured to compile bytecodes.

43. A single compiler system comprising:

means for using a compiler to obtain a machine-executable version of a virtual machine kernel; and

means for running the virtual machine kernel to provide a first virtual machine configured to perform mixed-mode execution of input program code using an interpreter and the compiler; and

means for providing the compiler to the first virtual machine as part of the input program code;

wherein mixed-mode execution comprises determining whether a current portion of input program code is to be compiled or interpreted, and wherein the current portion of input program code comprises a portion of the compiler.

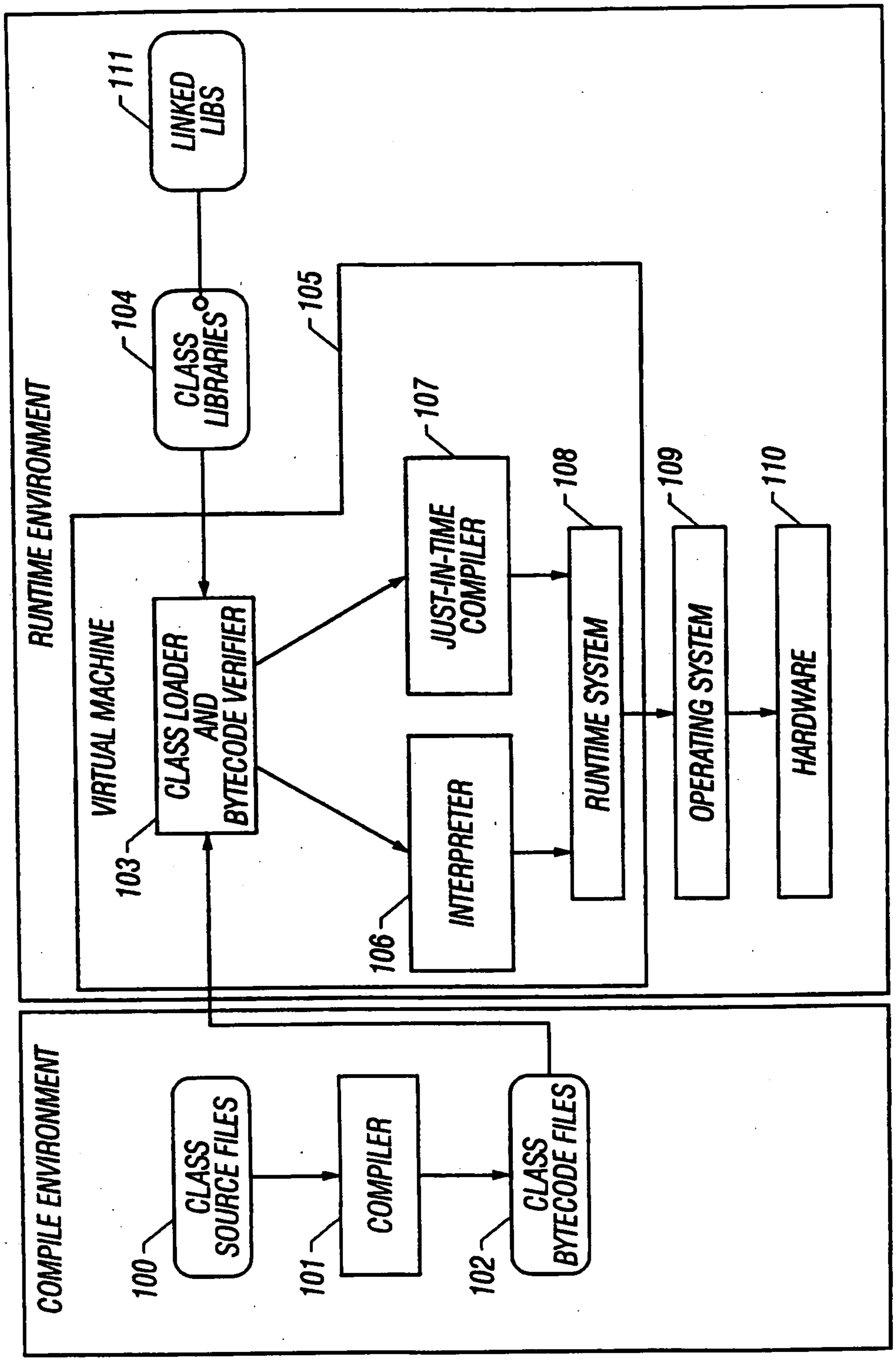
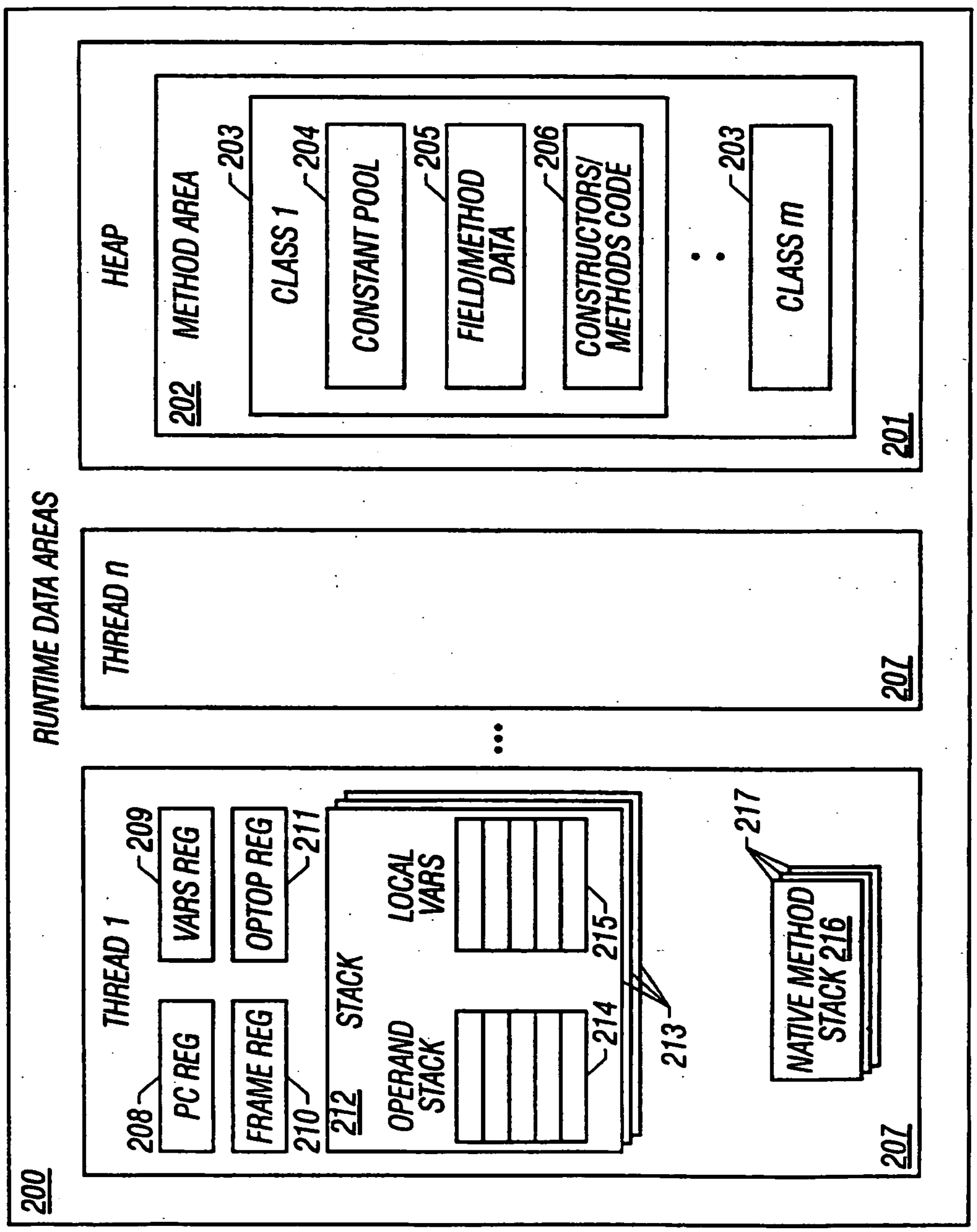
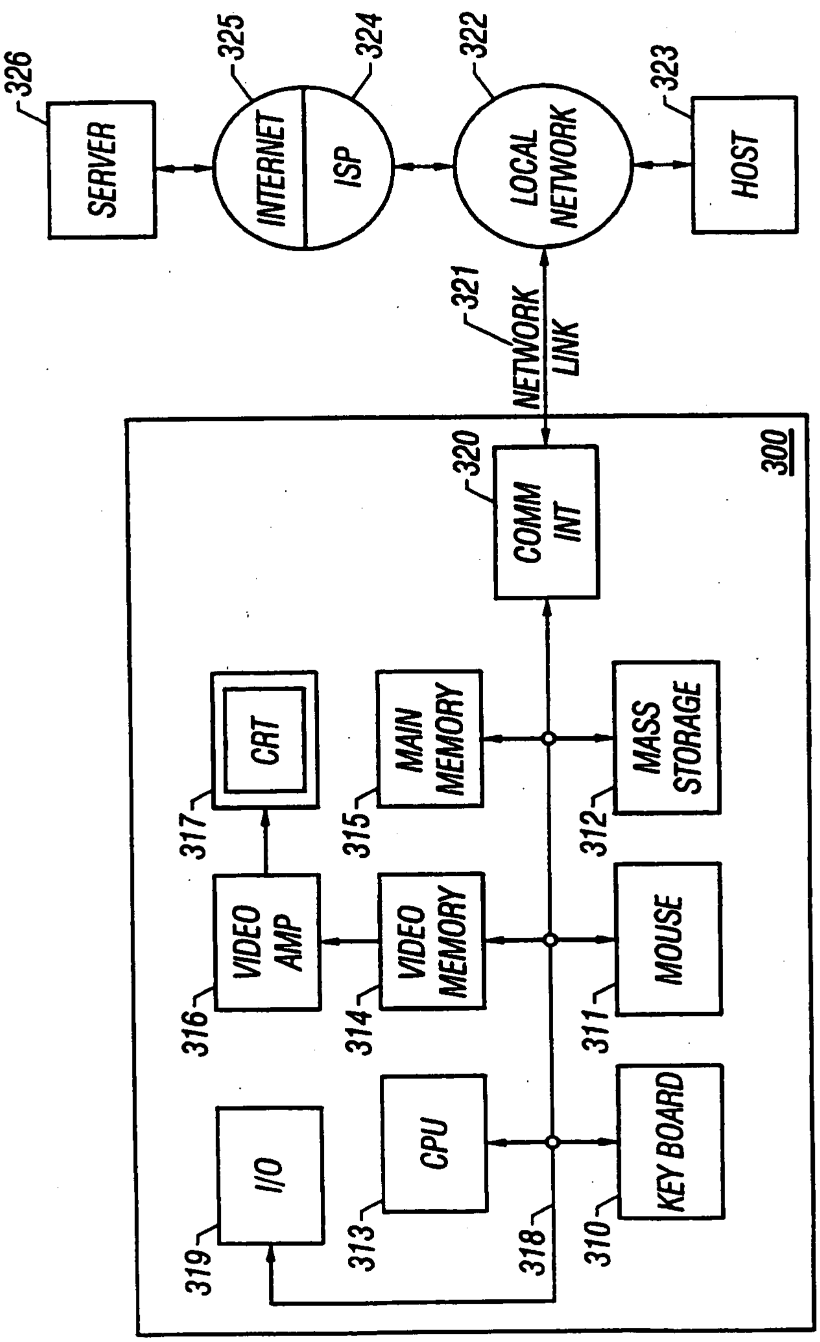

RUNTIME ENVIRONMENT
LINKED LIBS
111
CLASS LIBRARIES
104
105
VIRTUAL MACHINE
CLASS LOADER AND BYTECODE VERIFIER
103
JUST-IN-TIME COMPILER
107
INTERPRETER
106
RUNTIME SYSTEM
108
OPERATING SYSTEM
109
HARDWARE
110
COMPILE ENVIRONMENT
CLASS SOURCE FILES
100
COMPILER
101
CLASS BYTECODE FILES
102

FIGURE 1



200
RUNTIME DATA AREAS
HEAP
201
METHOD AREA
202
203
CLASS 1
204
CONSTANT POOL
205
FIELD/METHOD DATA
206
CONSTRUCTORS/ METHODS CODE
203
CLASS m
THREAD n
207
THREAD 1
207
208
PC REG
209
VARS REG
210
FRAME REG
211
OPTOP REG
212
STACK
OPERAND STACK
214
LOCAL VARS
215
213
217
NATIVE METHOD STACK 216

FIGURE 2



326
SERVER
325
INTERNET
ISP
324
322
LOCAL NETWORK
323
HOST
321
NETWORK LINK
320
COMM INT
300
317
CRT
315
MAIN MEMORY
312
MASS STORAGE
316
VIDEO AMP
314
VIDEO MEMORY
311
MOUSE
319
I/O
313
CPU
318
310
KEY BOARD

FIGURE 3

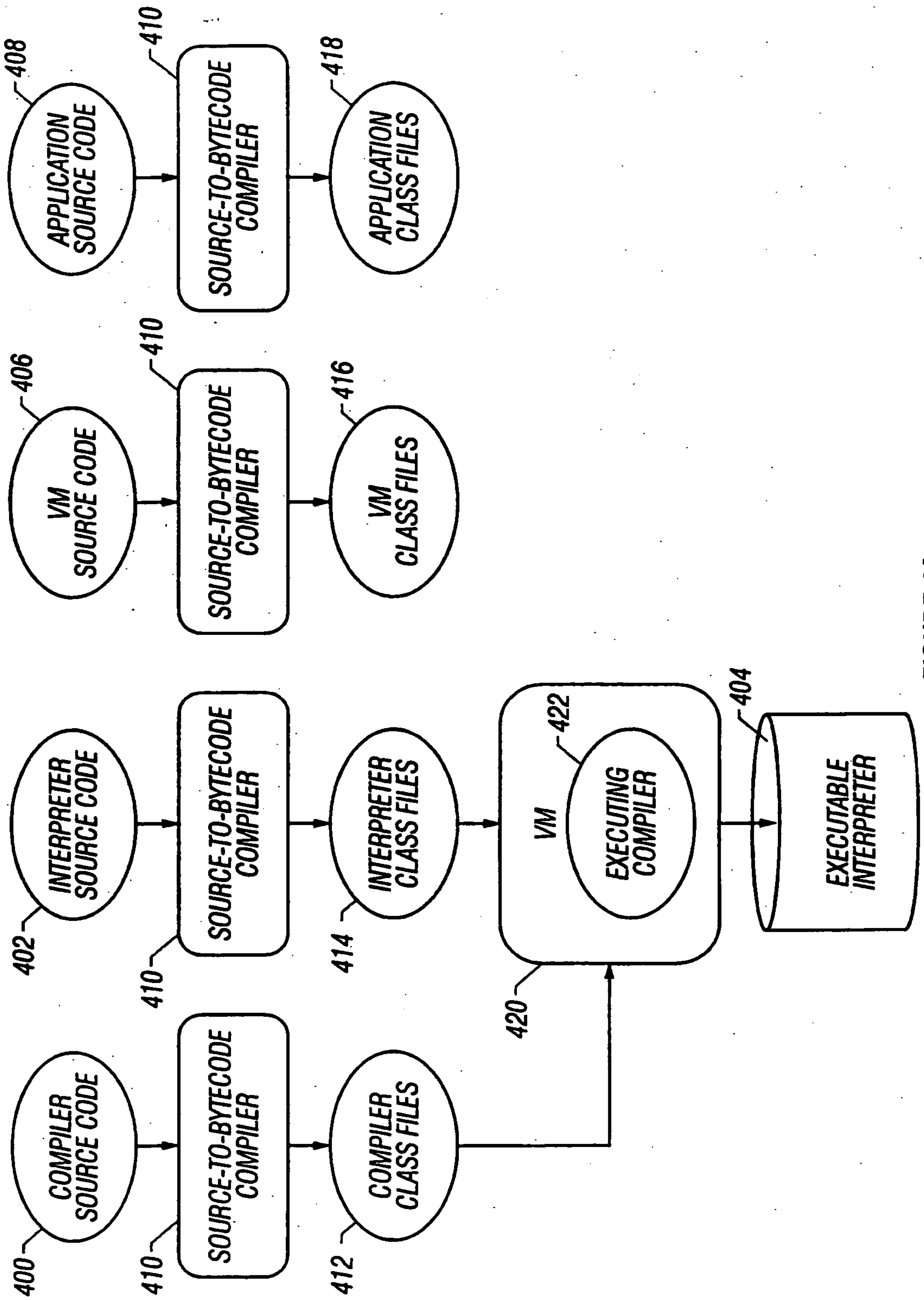

COMPILER SOURCE CODE
400
SOURCE-TO-BYTECODE COMPILER
410
COMPILER CLASS FILES
412
INTERPRETER SOURCE CODE
402
SOURCE-TO-BYTECODE COMPILER
410
INTERPRETER CLASS FILES
414
VM
420
EXECUTING COMPILER
422
EXECUTABLE INTERPRETER
404
VM SOURCE CODE
406
SOURCE-TO-BYTECODE COMPILER
410
VM CLASS FILES
416
APPLICATION SOURCE CODE
408
SOURCE-TO-BYTECODE COMPILER
410
APPLICATION CLASS FILES
418

FIGURE 4A

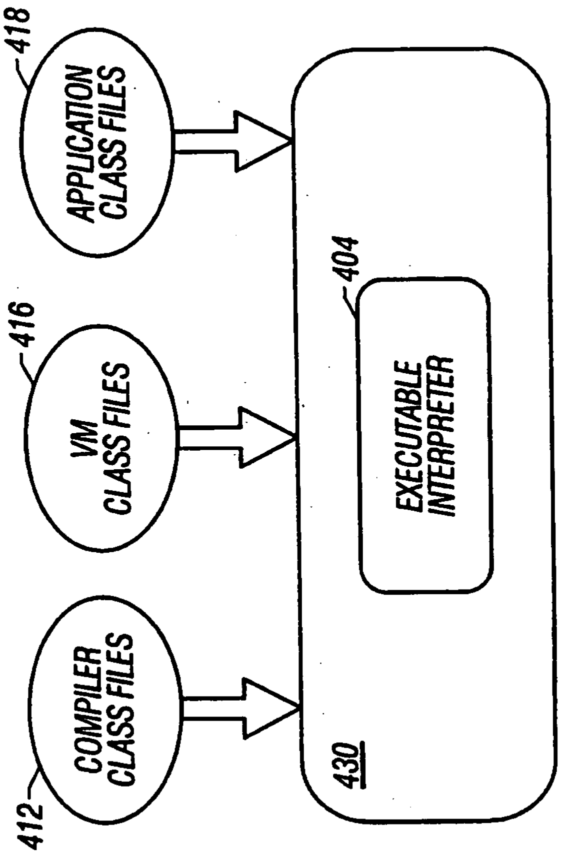


FIGURE 4B

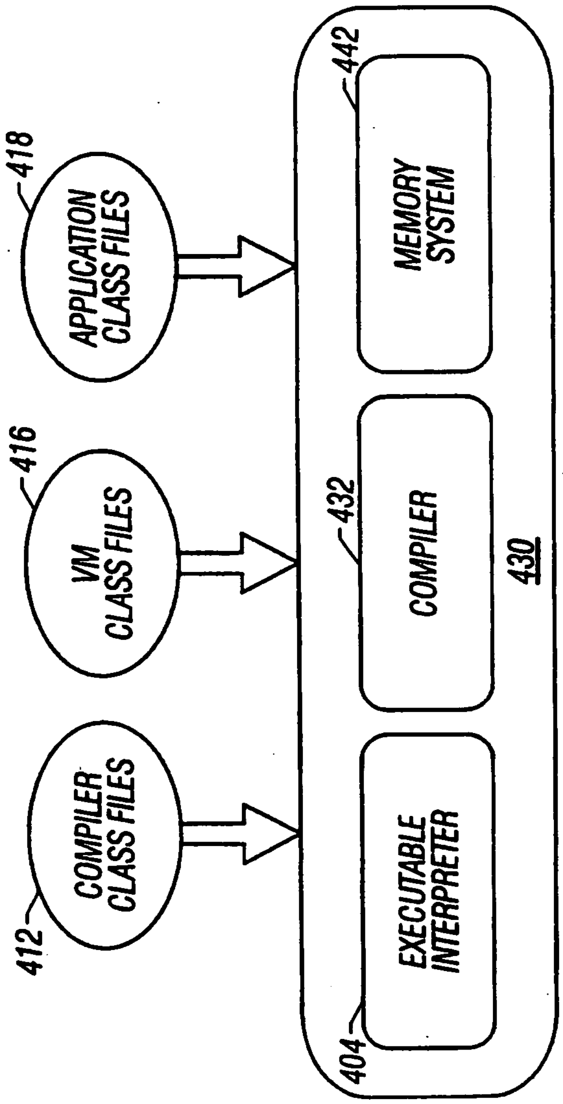


FIGURE 4C

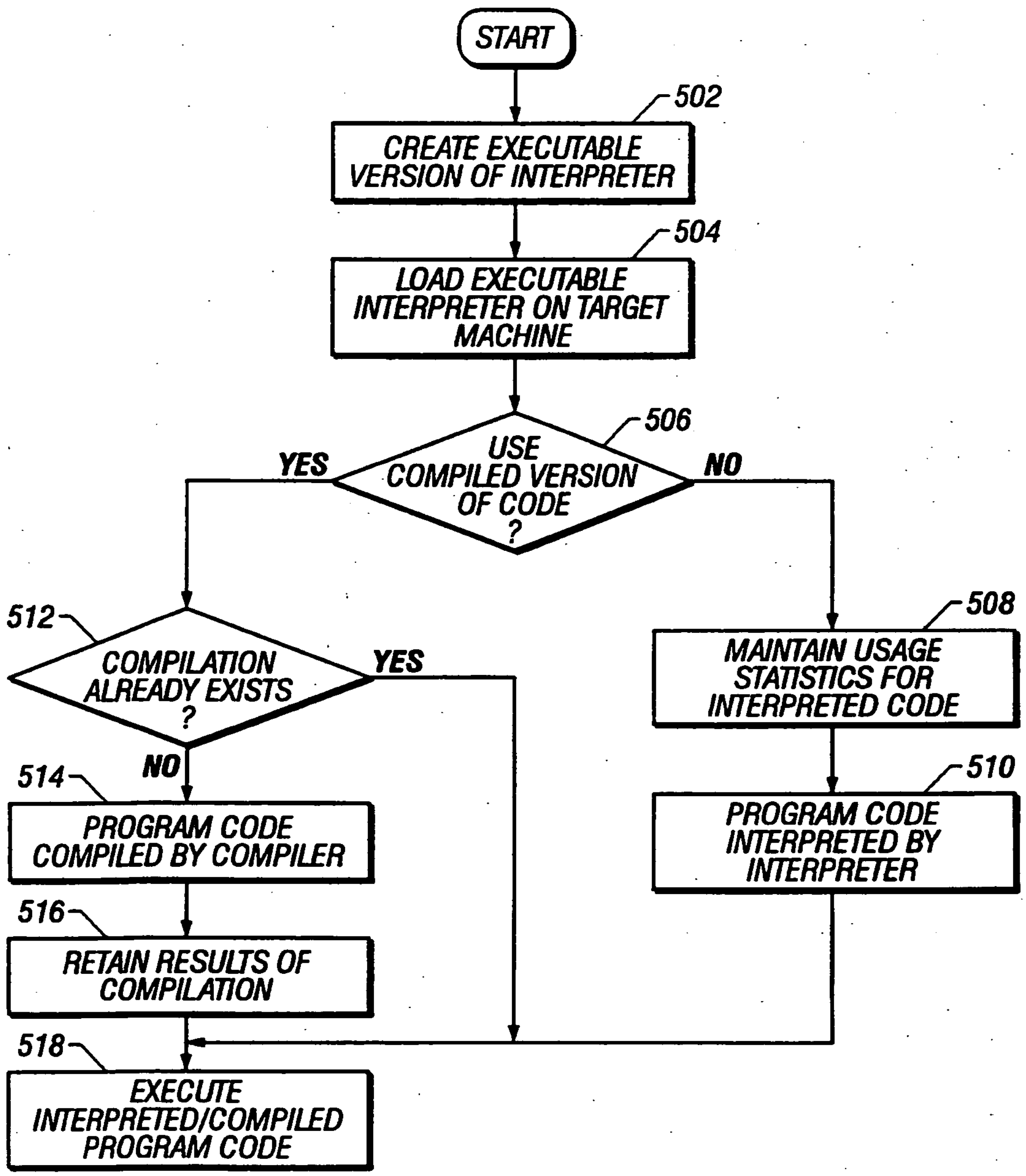

START
502
CREATE EXECUTABLE VERSION OF INTERPRETER
504
LOAD EXECUTABLE INTERPRETER ON TARGET MACHINE
506
USE COMPILED VERSION OF CODE ?
YES
NO
512
COMPILATION ALREADY EXISTS ?
YES
NO
508
MAINTAIN USAGE STATISTICS FOR INTERPRETED CODE
510
PROGRAM CODE INTERPRETED BY INTERPRETER
514
PROGRAM CODE COMPILED BY COMPILER
516
RETAIN RESULTS OF COMPILATION
518
EXECUTE INTERPRETED/COMPILED PROGRAM CODE

FIGURE 5

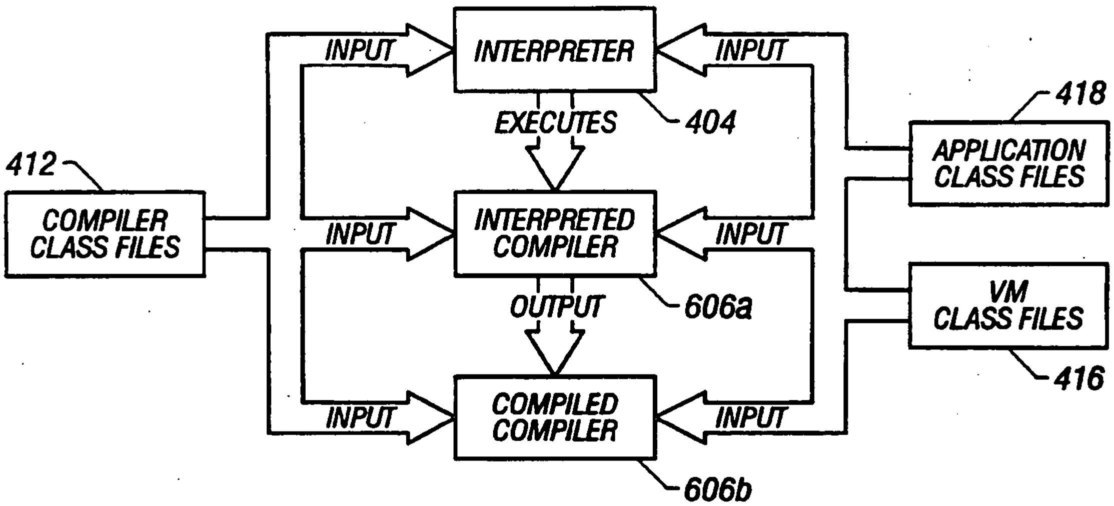


FIGURE 6

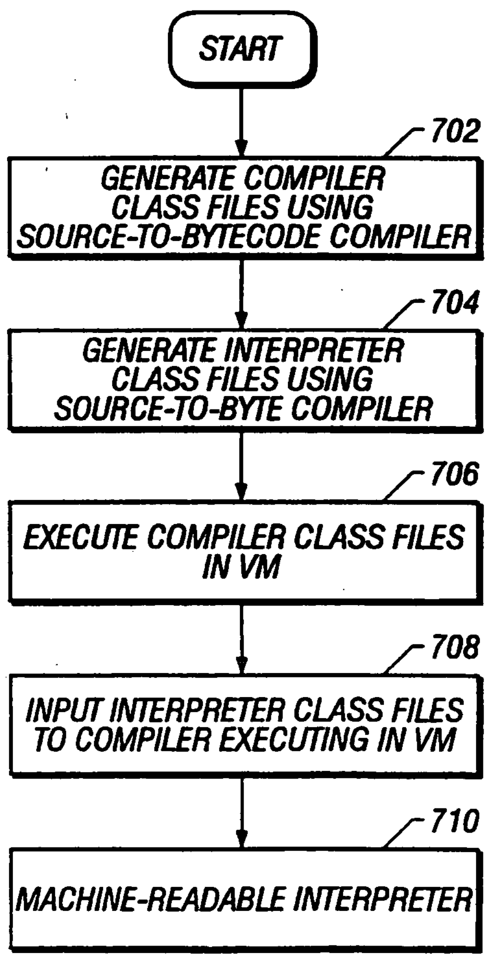


FIGURE 7

This Page Blank (uspto)